523

NPSCS-93-012

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Robust Scheduling for Large Projects

by Valdis Berzins
Salah Badr

December 1993

Approved for public release; distribution is unlimited.

Prepared for:

Army Research Office
Research Triangle Park, NC

FedDocs
D 208.14/2
NPS-CS-93-012

Fedlong
D 202.14/2
NPS-CS-93-012

NAVAL POSTGRADUATE SCHOOL
Monterey, California

REAR ADMIRAL T. A. MERCER
Superintendent

HARRISON SHULL
Provost

This report was prepared for and funded by the Army research Office under grant number ARO-145-91 and the National Science Foundation under grant number CCR-9058453.

Reproduction of all or part of this report is authorized.

This report was prepared by:

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY CA 93943-5101

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) NPSCS-93-012	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Computer Science Dept. Naval Postgraduate School 6b. OFFICE SYMBOL *(if applicable)* CS	7a. NAME OF MONITORING ORGANIZATION Army Research Office
6c. ADDRESS *(City, State, and ZIP Code)* Monterey, CA 93943	7b. ADDRESS *(City, State, and ZIP Code)* Research Triangle Park, NC
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Naval Postgraduate School 8b. OFFICE SYMBOL *(if applicable)* NPS	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER Grant number ARO-145-91
8c. ADDRESS *(City, State, and ZIP Code)* Monterey, CA 93943	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
Robust Scheduling for Large Projects

12. PERSONAL AUTHOR(S)
Valdis Berzins and Salah Badr

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Interim	FROM _____ TO _____	1993 December	22

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUB-GROUP	Project Scheduling, uncertainty, Automatic Deadline Adjustment

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

We have developed a heuristic scheduling method suitable for automatically scheduling tasks in a software development effort and assigning them to designers. Our experimental evaluations of the algorithm show that it is highly effective at finding feasible schedules when they exist. A modification of the algorithm can suggest nearly minimal adjustments to the deadlines in cases where no feasible schedule exists.

This is useful because it provides guidance to the project manager for formulating aproper response when a project gets late and all of the planned tasks cannot be completed with their deadlines. The algorithms are fast enough to support constant rescheduling as circumstances change, for most projects of practical size.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION	
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL Valdis Berzins	22b. TELEPHONE *(Include Area Code)* (408) 656-2461	22c. OFFICE SYMBOL CSBe

Robust Scheduling for Large Projects

Valdis Berzins
Salah Badr
Department of Computer Science
Naval Postgraduate School
Monterey, California 93943 USA
E-mail:berzins@cs.nps.navy.mil
badr@cs.nps.navy.mil

ABSTRACT

We have developed a heuristic scheduling method suitable for automatically scheduling tasks in a software development effort and assigning them to designers. Our experimental evaluations of the algorithm show that it is highly effective at finding feasible schedules when they exist. A modification of the algorithm can suggest nearly minimal adjustments to the deadlines in cases where no feasible schedule exists.

This is useful because it provides guidance to the project manager for formulating a proper response when a project gets late and all of the planned tasks cannot be completed within their deadlines. The algorithms are fast enough to support constant rescheduling as circumstances change, for most projects of practical size.

A. INTRODUCTION

We have investigated scheduling of designer tasks in the context of developing an automated evolution control system. The purpose of this system is to provide automated assistance for coordinating a team of designers in the context of software prototyping. The system also manages all software documents and ensures that each designer automatically receives all documents relevant to each work assignment.

In this context, development speed and flexibility are primary considerations, and the expected level of uncertainty is very high. Our results suggest that automated assistance for team coordination, scheduling, and assignment of tasks to designers is feasible and can be practically useful for realizing fast modifications to software designs.

This research was supported in part by the Army Research Office under grant number ARO-145-91 and the National Science Foundation under grant number CCR-9058453.

B. PREVIOUS WORK

A scheduling problem in a real-time system is described by three basic concepts: the model of the system, the characteristics of the tasks to be scheduled, and the objective of the scheduling algorithm [3].

The system model in our case consists of a set of m designers $D = \{d_1, d_2, .., d_m\}$. Those designers are of three different expertise levels {low, medium, high}. The scheduling algorithm determines the order of the execution of tasks by each designer in such a way that resource, precedence, and timing constraints are met. In our system resources required by a task other than the designer resources are assumed to be available as soon as the task is assigned.

The tasks to be scheduled, evolution steps in our case, are characterized by their timing constraints, precedence constraints, and resource requirements. The timing constraints of a task are generally defined in terms of one or more of the following parameters [3]:

1. The arrival time, T_a: The time at which a task arrives at the system.
2. The earliest start time, T_{est}: The earliest time at which a task can start execution. (invariant: $T_{est} \geq T_a$).
3. The worst case execution time, T_c: The execution time of a task is always less than T_c.
4. The deadline, T_d: The time by which a task must be completed.
   (invariant : $0 \leq T_a \leq T_{est} \leq T_d - T_c$)

While all the tasks and their timing constraints are known beforehand in a static system, tasks arrive at arbitrary times in a dynamic system, so that the number of tasks to be scheduled as well as their arrival times are unpredictable.

The relations between the tasks are determined by the ***precedence*** constraints among these tasks. If a task T_i must be completed before another task T_j can be started then we say T_i precedes T_j. The precedence graph of a set of tasks is a directed acyclic graph. This precedence graph is known in advance in static systems. In dynamic systems where new sets of interrelated tasks arrive arbitrarily, the precedence graph is known only when the task set arrives.

The objective of an algorithm for scheduling a set of tasks is to determine whether there exists a schedule for executing the tasks that satisfies the timing, precedence, and resource constraints, and to calculate such a schedule if one exists.

Task scheduling in real-time systems can be static or dynamic. A static approach performs the calculation of the schedules for tasks off-line. It requires prior knowledge of the characteristics of the tasks. On the other hand, a dynamic approach calculates schedules for tasks "on the fly". Despite the fact that static approaches have low run-time cost, they are inflexible and cannot respond to a changing environment with unpredictable behavior. In contrast, dynamic approaches involve higher run-time costs, but they are flexible to adapt to environment changes. A survey of static and dynamic scheduling approaches can be found in [3].

Task scheduling can also be characterized as preemptive and nonpreemptive. A task is preemptive if its execution can be interrupted by other tasks and resumed afterwards. A task is nonpreemptive if it must run to completion once it starts.

1. Scheduling Tasks with Precedence Constraints

Scheduling tasks with arbitrary precedence constraints and unit computation time in multiprocessor systems is NP-hard for both the preemptive and nonpreemptive cases [3] [6]. Scheduling nonpreemptive tasks with arbitrary ready times is NP-hard in both multiprocessor and uniprocessor systems [3] [5] which excludes the possibility of the existence of a polynomial time algorithm for solving the problem. Hong and Leung [2] proved that there is no optimal on-line scheduler can exist for task systems that have two or more distinct deadlines when scheduled on m identical processors where $m > 1$.

Scheduling evolution steps to more than one designer with arbitrary precedence constraints and arbitrary deadlines is the same problem as that of multiprocessor scheduling mentioned above which is shown by many researchers to be NP-hard. These negative results dictate the need for heuristic approaches to solve scheduling problems in such systems.

In [4] Stankovic et al. present an $O(n^2)$ heuristic scheduling algorithm for scheduling a set of independent processes on a set of identical processors. A task (process)

in this model is characterized by an arrival time T_A, a deadline T_D, a worst case computation time T_C, and a set of resource requirements $\{T_R\}$. Tasks are independent, non periodic and non-preemptive. The authors stated that scheduling a set of tasks to find a full feasible schedule is a search problem with a search tree as the search space. The scheduling algorithm starts at the root of the tree which is an empty schedule. It tries to extend the schedule by moving to the one of the nodes in the next level of the search tree until it reaches a full feasible schedule. During the expansion of the schedule, an intermediate node is a partial schedule, while leaf nodes (terminal node) represent full schedules. It is clear that not every terminal node corresponds to a feasible schedule. To extend the schedule to a node of the next level of the search tree, the algorithm uses a boolean function called "strongly-feasible" to determine if the partial schedule can lead to a feasible schedule or not. A partial schedule is strongly-feasible if all schedules reached by extending it by each of the remaining tasks are also feasible. This means that if a partial feasible schedule is found not to be strongly-feasible because a task T misses its deadline, then the search should stop on this path since none of the future extensions of task T can meet its deadline. However, it is possible to backtrack to continue the search in such cases. After deciding that a partial schedule is strongly-feasible, a heuristic function (H) is used to direct the search to a plausible path.

This algorithm works as follows: Given a particular heuristic function H, the algorithm begins with an empty partial schedule. Every step of the algorithm includes (a) determining if the current partial schedule is strongly-feasible, and if so (b) extending the current partial schedule by one task. This task is selected by applying the H function to all the tasks remaining to be scheduled and determining the one with the minimum H value.

Some of the H functions used in [4] are Minimum deadline first(Min_D), Minimum processing time first (Min_P), Minimum earliest start time first (Min_S), Minimum laxity first (Min_L), and the combinations (Min_D + Min_P) and (Min_D + Min_S).

In [3], Ramamritham et al. introduce an O(nk) version of the algorithm introduced in [4] by considering only k tasks of the remaining tasks to be scheduled for applying the H function and evaluating the strongly-feasible function.

Both [4] and [3] use a vector data structure for each type of resource to maintain the earliest available time for each resource of each type. In our algorithm for scheduling evolution steps we extend this algorithm to handle the case where there are precedence constraints between pairs of steps, and keep a vector of earliest available times of designers for each expertise level.

C. PROBLEM DEFINITION

Our problem is to schedule a set of sporadic tasks (software evolution steps). These sporadic tasks have random arrival times, and given deadlines, precedence constraints, and priority values to indicate the criticalness of their deadlines. Because of the unpredictable nature of the arrival time of the sporadic tasks, it is very difficult to design a real-time (on-line) system that guarantees that all their deadlines can be met [2]. Moreover, each of these tasks requires certain expertise level, which implies that the system model is a set of M software designers of different expertise levels (not identical designers). This problem is similar to that of dynamic scheduling tasks with arbitrary arrival times, deadlines, and precedence constraints in a multiprocessor system where the processors are not identical.

Hong and Leung [2] proved that there is no optimal on-line scheduler can exist for task systems that have two or more distinct deadlines when scheduled on m identical processors, where $m > 1$. Scheduling tasks with arbitrary precedence constraints and unit computation time is NP-hard both the preemptive and the non-preemptive case [3]. Our problem is even more complicated than both of the above two cases, when contrasted with the case proven in [2] we have more than one designer and each step of the step set has its distinct deadline which is the same conditions for the conclusion reached by Hong and Leung, in addition, the designers are not of the same expertise level which makes it even more complicated. In contrast with results of [3] our problem includes arbitrary precedence constraints between pairs of the steps in the step set to be scheduled in addition to an arbitrary computation time for each step which makes it even harder than the case of having unit computation time.

These negative results indicate the need for heuristic approaches solve this scheduling problem.

1. The Scheduling Algorithm

Scheduling a set of tasks to reach a feasible schedule is a search problem, where the search space can be structured as a search tree. The root of this search tree is an empty schedule, an intermediate node is a partial schedule, and a leaf node (terminal) is a complete schedule. Since not all leaves correspond to feasible schedules, it might cause an exhaustive search to find one, which is computationally intractable in the worst case. Because of the computational complexity of finding a full feasible schedule in many of the real applications, heuristic approaches are used.

a. System and Task Model

The task set in the ECS scheduling problem is a variable set of evolution steps $S = \{S_1, S_2,..., S_N\}$, where N varies with time. This set of tasks need to be scheduled to a set of M designers $D = \{D_1, D_2,..., D_M\}$. The designers are of L different expertise levels. Tasks (steps) are characterized by the following:

- Estimated processing time tp (S_i): a management estimate of the time required to perform a step.
- Deadline d (S_i): The time by which a step must be completed
- Earliest start time EST (S_i): the earliest time at which the step can be assigned to a designer (calculated when a scheduling decision is made).
- Priority p (S_i): An integer value to reflect the criticalness of the deadline of a step.
- Resource requirement r (S_i): required expertise level for performing a step.
- Precedence constraints given in the form of a directed acyclic graph G = (S, E) such that $(S_i, S_j) \in E$ implies that S_j cannot start until S_i has been completed.

In order to support teamwork, we assume that each step is assigned to a single designer. This designer must have at least the same expertise level as that of the step. We also define the earliest start time EST (Si) as the earliest time at which the step can be assigned to a designer. This time is calculated when a scheduling decision is made.

Our goal is to determine whether there exists a schedule for executing the tasks, that satisfies the timing, precedence, and resource constraints, and to calculate such

a schedule if one exists. Since this problem is computationally intractable, we weaken the requirements to checking whether a feasible schedule can be found within the available time. Otherwise the program should advise the software manager of the lowest priority deadlines that have to be canceled (moved to their calculated finish time) in order to get a feasible schedule. This algorithm should also give the software manager the choice to change other constraints such as priority, precedence or estimated execution time of the tasks to tune the schedule each time new evolution steps are to be added to the schedule and a feasible schedule cannot be reached. It also must check the validity of these changes (e.g. if a priority of a step is changed it has to be less than or equal the priorities of its predecessors and greater than or equal to that of its successors).

Thus, we need an on-line scheduler that is called when one or more sporadic tasks arrive at time t (new tasks in our system may have some of the constraints not defined when they arrive to the scheduler) or if the attributes of the currently scheduled tasks change, to decide if the newly arrived tasks, or the changed tasks, along with unassigned tasks at time t (scheduled but not started yet), could be rescheduled so that all deadlines are met. If a feasible schedule is reached the system will continue assigning the tasks to the designers according to the schedule constructed by the on-line scheduler. Otherwise the system will try to meet the deadlines of the most important (highest priority) tasks and suggest changing the deadlines of the least important ones. These suggestions could be accepted by the manager or he can change other parameters which in turn triggers the on-line scheduler to recalculate the schedule accordingly.

Changing the attributes of currently scheduled tasks means editing any of the constraints of the not-started-yet tasks, assigned tasks that are prone to exceed their estimated execution time (which is a common case in software effort estimation), and the addition/deletion of designers.

D. PROBLEM SOLUTION

A heuristic scheduling algorithm tries to reach a feasible schedule for a set of tasks by starting at the root of the search tree, which is an empty schedule, and tries to extend the schedule with one more task by moving to one of the nodes in the next level of the search

tree until a feasible schedule is reached. The nodes in the next level of the search tree consist of those tasks that are ready to be scheduled, i.e. the tasks that have all their predecessors completed at this point or has no predecessors. A partial search path is extended only if it is strongly feasible.This is because if extending the current schedule by a task T causes T to miss its deadline then none of all the possible future extensions can meet the deadline of task T, since starting T later cannot make it finish earlier [3]. To this point we introduce the following definition:

- Strongly-feasible partial schedule: A partial schedule is strongly-feasible if all schedules reached by extending it by any of the remaining (ready to be scheduled) tasks are also feasible.

If the partial schedule is strongly feasible then a heuristic function is used to extend the partial schedule. This heuristic function should reflect various characteristics of the scheduling problem to effectively direct the search to a plausible path. If all the schedules resulting from extending the current schedule with any of the remaining tasks are also feasible, the partial schedule is called strongly feasible. The heuristic function is then applied to every task that is ready to be scheduled. The task with a predefined property of the heuristic function is selected to extend the current partial schedule (e.g. if we use the earliest deadline first as our heuristic then we pick the task with earliest deadline of the tasks that are ready to be scheduled to extend the current partial schedule), otherwise this search path is stopped because it will not lead to a feasible schedule.

Our heuristic algorithm is based on the heuristic algorithm introduced in [3] and discussed above. The main difference is that the tasks in our problem have precedence constraints which is not discussed in [3] where the authors deal with a set of independent tasks. Another difference is that each task has its own deadline rather than a common deadline for each set of tasks as is the case in [3].

Before describing the details of our algorithm, let us introduce the following definitions:

- Pending_step: a step whose predecessors (in the dependency graph) have all been scheduled (not necessarily assigned yet) and their estimated finish time is calculated. The step's earliest start time is set to the latest finish time of its predecessors.

- Ready_step: a pending step whose earliest start time is less than or equal the current time t.

The following data structures and variables are used by the algorithm:

- Dependency_graph: a directed acyclic graph G = (S, E) such that S = $\{S_1, S_2,..., S_N\}$ is the set of steps to be scheduled, E is the set of edges such that $(S_i, S_j) \in E$ if and only if S_j cannot start until S_i has completed.
- In_degree: an integer representing the number of the immediate predecessors of each node (step) in the dependency graph.
- Pending_list: a list holding pending steps sorted in a non-decreasing order of their earliest start time.
- Ready_list: a list holding ready steps sorted in a non-decreasing order of the heuristic function used (e. g., deadlines, earliest start time etc.).
- Earliest Available Time (EAT): a vector of M values to represent the earliest available times of the resources (designers). EAT_i is the earliest time when D_i becomes available when the system has only one instance of each resource type (expertise level), e. g., for the case of having only three expertise level low, medium, and high and one designer of each level then EAT = (EAT_l EAT_m EAT_h). In case of having multiple instances of each expertise level the EAT is represented as a matrix so that each row represents the Earliest Available Times of the different instances of each expertise level.

$$EAT = ((EAT_{l1}\ EAT_{l2}\ ..\ EAT_{lk})$$

$$(EAT_{m1}\ EAT_{m2}\ ..\ EAT_{mr})$$

$$(EAT_{h1}\ EAT_{h2}\ ..\ EAT_{hp}))$$

where l, m, h are the three expertise levels low, medium, and high respectively, and k, r, and p are the corresponding number of designers in each level.

The main idea of this algorithm is to extend the current schedule by one of the steps in the ready list. The tasks in the ready list can be seen as independent tasks if we can define an earliest start time and a deadline for each of them. This is done for the deadlines by propagating them from the terminal to the root nodes in the dependency graph.

The propagated deadline d'(Si) of a step Si is defined by:

1) $d'(Si) = d(Si)$ if $\neg\exists$ Sj : Si precedes Sj

or

2) $d'(Si) = \min \{d(Si), d'(Sj) - tp(Sj)\}\ \forall$ Sj : Si precedes Sj

In 2) above, if there exists some step Sj such that Si precedes Sj then Sj cannot start until Si has completed. In order to complete Sj's computation before its deadline, the latest time by which Sj must be started is d'(Sj) - tp (Sj). Then Si's real deadline should be d'(Sj) - tp (Sj) if it is smaller than d (Si).

As for the earliest start time (EST) of each step, it is adjusted according to the following:

1) EST' (Si) = EST (Si) if $\neg\exists$ Sj : Sj precedes Si

or

2) EST' (Si) = max {EST (Si), EST'(Sj) + tp (Sj)} $\forall$ Sj : Sj precedes Si

In 2) above, if there exists some step Sj such that Sj precedes Si then Si cannot start until Sj has completed. Since the earliest time that Sj can be completed is EST'(Sj) + tp (Sj) then Si's real EST should be EST'(Sj) + tp (Sj) if it is greater than EST (Si).

The reason for having a pending_list and a ready_list instead of having one ready_list is to give the available tasks (in_degree = 0 and EST <= current time) a fair chance to compete for available designers especially when using different heuristics other than EST first, since the scheduler considers only the steps in the ready_list.

Our scheduling algorithm has two different initialization procedures. The first one is used when the system starts from scratch (i.e., the schedule is empty), while the second initialization procedure is used when new tasks arrive at the system or some of the attributes of an existing step is changed. This scheduling algorithm is similar to the branch and bound technique. The strong feasibility check done before extending the schedule by another node in the search tree is used instead of the lower bound check, normally used with branch and bound algorithm, to bound the search in a given search path. The algorithm works as follows:

Initialization_part:

(1) if initial_schedule = empty

(2) then

initialize EAT values to T0, and the schedule to empty.
perform a Depth First Search on the dependency graph to:

- initialize the in_degree for each node (number of immediate predecessors),
- propagate deadlines, and
- initialize the ESTs (earliest start time) of the steps that have no EST to T0.

insert each pending step (its in_degree = 0) into the pending list ordered by its EST.

(3) else

update the dependency_graph:

- Remove the assigned steps and their corresponding arcs from the dependency graph
- Add the newly arrived steps to the dependency graph (if there is any) checking for the "acyclic" property of the graph and the compatibility of the newly added steps' priorities with that of their successors and predecessors and warn the manager of any violation

Recalculate the in_degree of the graph nodes.

Re-initialize the EAT vector (matrix) to the finish time of the step assigned to each designer and to t for the free designers.

Insert each pending step (its in_degree = 0) into the pending list ordered by its EST.

end if

Schedule_part:

(4) While full_schedule is not reached loop

(5) For all the steps in the pending list:

if EST (S) <= min(EAT) of the corresponding designers then

insert S into the ready_list in order of non-decreasing values of the H (heuristic) function used and delete S from the pending list.

(6) end for

(7) While ready_list is not empty loop

(8) if not STRONGLY_FEASIBLE to extend the schedule by each of the steps in the ready_list then

if the backtrack limit is not reached then increment backtrack counter and backtrack (discard the current partial schedule and backtrack to the previous partial schedule and extend it by a different step)

else exit (NO_FEASIBLE_SCHEDULE)

end if

end if

(9) extend the schedule by the step S that has min H

in case of ties, select the step Si with the highest priority, then the step with max tp(Si)

```
(10)          update the EAT of the assigned designer
(11)          update the EST of the immediate successors of S
(12)          decrement the in_degree of the immediate successors of S
(13)          if the in_degree of any of the immediate successors of S = 0
              then
                  insert it into the pending_list in order of its EST,
              end if.
(14)          delete S from the ready_list
(15)    end while
(16) end while
```

The STRONGLY_FEASIBLE is a boolean function that works as follows:

```
FEASIBLE = TRUE
for all the steps S in the ready_list loop
    if min (EAT) of the designers of the same or higher expertise level than
    level(S) + Estimated_duration(S) > deadline(S)
    then FEASIBLE = FALSE
    end if
end for
```

The following are some of the heuristics that may be used with this algorithm:

- Minimum deadline first (Min_d): H(S) = d (S)
- Minimum earliest start time first (Min_est): H(S) = EST (S)
- Minimum laxity first (Min_L): H(S) = d (S) - (EST (S) + tp (S))
- Min_d + Min_est first: H(S) = W * d (S) + (1-W) * EST (S)
- In the four cases ties are broken using the priorities of the steps (the highest priority step starts first). Further ties are broken by selecting the step that has the maximum tp.

The first three heuristics are simple heuristics and the last one is an integrated heuristic. The weight W (0 <= W <= 1), used to combine the two simple heuristics Min_d and Min_est, can be tuned according to the criticalness of the deadlines of the available steps. This means if the deadlines are not critical then W can be set to 0 which leads to Min_est heuristic that is the best for team work to assign tasks to designers according to their earliest start time making a full use of the human resources. On the other hand the

value of W can be chosen to favor the deadline heuristic or some way in between to meet the critical deadlines and make the best use of the human resources (designers) available.

The backtracking limit is left open in the cases where the number of tasks is relatively small, and is limited otherwise. In the cases where no feasible schedule is reached either due to the absence of a feasible schedule for the given set of tasks or due to reaching the backtracking limit of the algorithm without reaching one, an algorithm for adjusting the deadlines is used. This enhancement to the algorithm is presented in the next section. This valid schedule can be improved on by using the simulated annealing technique.

1. Algorithm for Adjusting Deadlines

A valid schedule is a schedule that satisfies the precedence constraints of its tasks but allows some of the tasks to miss its deadlines. Different heuristics can be used to guide the search process to a plausible path that minimizes the number of tasks that must miss its deadlines and in the mean time supports team work by scheduling every available task as soon as the earliest available time of the task is reached. This in turn minimizes the time a designer has to wait for a task to be assigned to him/her.

This algorithm uses almost the same steps as in the previous search algorithm uses with two main differences. The first difference is that: there is one ready_lists for each of the L expertise levels. The main reason for having the different levels of ready_lists is to guarantee that no lower task is assigned to a higher level designer while there is a task of the designer's level ready to be assigned (recall the requirement that the expertise level of the designer must be at least the same as that of the assigned task). The second difference is that when failing the strong feasibility check for extending the schedule by another task, a new deadline is suggested for the task that does not meet its deadline (equal to its calculated finish time). Upon accepting this value by the manager the schedule is extended to the next level and so on until a valid schedule is reached.

The Proposed deadline-adjusting scheduling algorithm works as follows:

initialization_part:

(1) if initial_schedule = empty
(2) then

initialize EAT values to T0, and the schedule to empty.

perform a Depth First Search on the dependency graph to:

- initialize the in_degree for each node (number of immediate predecessors),
- propagate deadlines, and
- initialize the ESTs (earliest start time) of the steps that have no EST to T0.

Insert each pending step (its in_degree = 0) into the pending list according to its EST.

(3) else

update the dependency_graph:

- Remove the assigned steps and their corresponding arcs from the dependency graph.
- Add the newly arrived steps to the dependency graph (if there is any) checking for the "acyclic" property of the graph and the compatibility of the newly added steps' priorities with that of their successors and predecessors and warn the manager of any violation.

Recalculate the in_degree of the graph nodes.

Re-initialize the EAT vector (matrix) to the finish time of the step assigned to each designer and to t for those free designers.

Insert each pending step (its in_degree = 0) into the pending list ordered by its EST.

end if

schedule_part:

(4) While full_schedule is not reached loop

(5) For all the steps in the pending list:

if EST (S) <= min(EAT) of the corresponding designers then

insert the step into the corresponding ready_list according to the H (heuristic) function used and delete it from the pending list.

end if

(6) end for

(7) **For all ready_lists from higher_level to lower_level loop**

(8) While ready_list is not empty loop

(9) if not FEASIBLE to extend the schedule by any of the steps in the ready_list

then **suggest a new deadline for the infeasible step assignment**

if the suggestion is not accepted then exit, end if.

end if

(10) extend the schedule by the step S that has min H

(11) update the EAT of the assigned designer

```
(12)                update the EST of the immediate successors of S
(13)                decrement the in_degree of the immediate successors of S
(14)                if the in_degree of any of the immediate successors of S = 0
                    then
                        insert it into the pending_list,
                    end if.
(15)                delete S from the ready_list
                    T0 = min (EAT) of the designers of the same or higher expertise
                        level than level(ready_list)
(16)                for all the steps S in the pending list such that expertise_level (S) =
                        level (ready_list):
                        if EST (S) <= T0
                        then
                            insert S into the ready_list according to the H function used
                            and delete it from the pending list.
                        end if
                    end for
                end while
                if not FEASIBLE then exit end if
(18) end for
                if not FEASIBLE then exit end if
(19) end while
```

This algorithm has the property that a designer will never be left idle when there is a ready step that the designer is qualified to do. This is because inserting steps into ready list and their assignment to designers are triggered by the availability of designers as is the case in statement 5, 10, and 15.

2. Complexity Analysis

Both of the two algorithms introduced above have a total of n steps, where n is the number of the tasks to be scheduled. The complexity of each step is determined by the complexity of the computation done to determine strong feasibility and the complexity of H function evaluation. The strong feasibility calculation is linearly proportional to the number of the steps in the ready list. This number depends on the connectivity of the dependency graph which is n in the worst case. The H function computation is done simply

by inserting the ready steps into the ready list(s) in order of their H function which has the order of (log n) in the worst case if we use a heap data structure for the ready lists.

The overall worst case complexity of the algorithm is:

$n + (n - 1) + (n - 2) + .. +2 = O(n^2)$.

The backtracking of the first algorithm can be limited to a constant number which does not affect the complexity analysis. In our expermental results we found out that the backtracking number needed to schedule all of the feasible problems with tight deadlines in our statistical samples is at most proportional to n with a small constant (0.57) which leads to a worst case complexity of $O(n^3)$. It is also worth noting that the number of steps in the ready_list is linearly proportional to the remaining ready unassigned steps which is always less than or equal to the number of the remaining unassigned steps, so that the average case is expected to be much smaller than the worst case.

3. SIMULATION STUDY

The main goal of a scheduling algorithm is to find a feasible schedule for a set of tasks, if one exists. Clearly, a heuristic scheduling algorithm is not guaranteed to reach such a schedule. However, one heuristic algorithm is favored over another, if we have a number of task sets that known to have feasible schedules, the first is able to find feasible schedules for more task sets than the second. To take this approach, we have to come up with a number of task sets, each of which is known to have a feasible schedule. Unfortunately, only an exhaustive algorithm can find out whether an arbitrary task set can be feasibly scheduled.

Given m different designers, the complexity of an exhaustive search to find a feasible schedule for n tasks in the worst case can be $O(m^n * n!)$. This is why we take the approach taken by Ramamritham et. al. [3] which is using a task generator that can generate schedulable task sets where the number of tasks in each set can be arbitrarily large without adding much complexity on the task generator. Additionally, the tasks are generated to guarantee the total utilization of the available designers. These task sets are then input to

the scheduling algorithm, that has no knowledge that these sets are schedulable. The parameters used to generate task sets are:

1. The minimum duration of a task, Min_D.
2. The maximum duration of a task, Max_D.
3. The schedule length, L.

The task set generator starts with an empty EAT matrix, it then generates a task by selecting one of the n designers that have the earliest available time and then randomly chooses the task duration between the minimum duration and the maximum duration. The task generator then increments the EAT of the selected designer by the value of the task duration. The task generator generates tasks until the remaining unused time units of each designer, up to the schedule length L, is less than the minimum duration of a task, that means no more tasks can be generated for this designer within the given schedule length.

The deadline for each task is chosen randomly between the task's shortest completion time T_{sc} and (1+F) * T_{sc}, where F is a parameter indicating the tightness of the deadlines, and is related to the loading factor of each set of designers of the same expertise level. If F is 0, the scheduler must be able to find the same schedule as that found by the task generator in order to reach a feasible schedule. As the value of F is increased it is obvious that the scheduler has a better chance to find a feasible schedule for the task set.

a. Simulation Method

In our simulation study, N task sets are generated, where each set is known to be schedulable according to the task set generation procedure discussed above. Performance of different heuristics are compared according to how many of the N feasible task sets are found schedulable when the heuristics are used [3]. We use the same metric used in [3] which is defined as:

$SR = \frac{s}{N}$, where s is the number of schedulable task sets found by the heuristic algorithm, and N is the total number of task sets.

The loading factor for the designers is different according to their expertise level. we assume that the designers are of three different expertise levels High, medium and

low, and a step can be assigned to a designer that has at least the same expertise level as that required by the step. This assumption makes the loading factor vary for the designers in different levels as defined below.

For high level designers we define the loading factor as follows:

$$LFh = \frac{\sum Tph}{(Max(di) - To) \times Mh}$$

where LFh is the loading factor for high level designers, Tph is the estimated duration for a high level task, To is the initial start time for scheduling the tasks, Mh is the number of available high level designers and di is the deadline of task i.

For a medium level designer we define the loading factor as follows:

$$LFm = \frac{\sum Tpm}{(Max(di) - To) \times (Nm + Nh - Nh \times LFh)}$$

$$LFm = \frac{\sum Tpm}{(Max(di) - To) \times Mm + (1 - LFh)\ (Max(di) - To) \times Mh}$$

where LFm is the loading factor for medium level designers, Tpm is the estimated duration for a medium level task and Mm is the number of available medium level designers.

For a low level designer we define the loading factor as follows:

$$LFl \frac{\sum Tpl}{(Max(di) - To) \times Nl + (1 - LFm)\ (Max(di) - To) \times (Nm + Nh - Nh \times LFh)}$$

$$LFl = \frac{\sum Tpl}{(Max(di) - To)\ (N - Nh(LFh + LFm - LFh \times LFm) - LFm \times Nm)}$$

where LFl is the loading factor for low level designers, Tpl is the estimated duration for a low level task and Ml is the number of available low level designers.

b. Simulation Results.

TABLE 1. Relation between Success Ratio (SR) and Laxity (L)

Laxity (F)	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7
Heuristic Search	100	100	100	100	100	100	100	100
Min_D	6	14	14	40	70	72	86	100
Min_s	0	0	0	0	8	10	10	22
Min_D + Min_S	0	0	0	0	0	0	8	16
Min_L	0	0	0	0	0	0	8	10

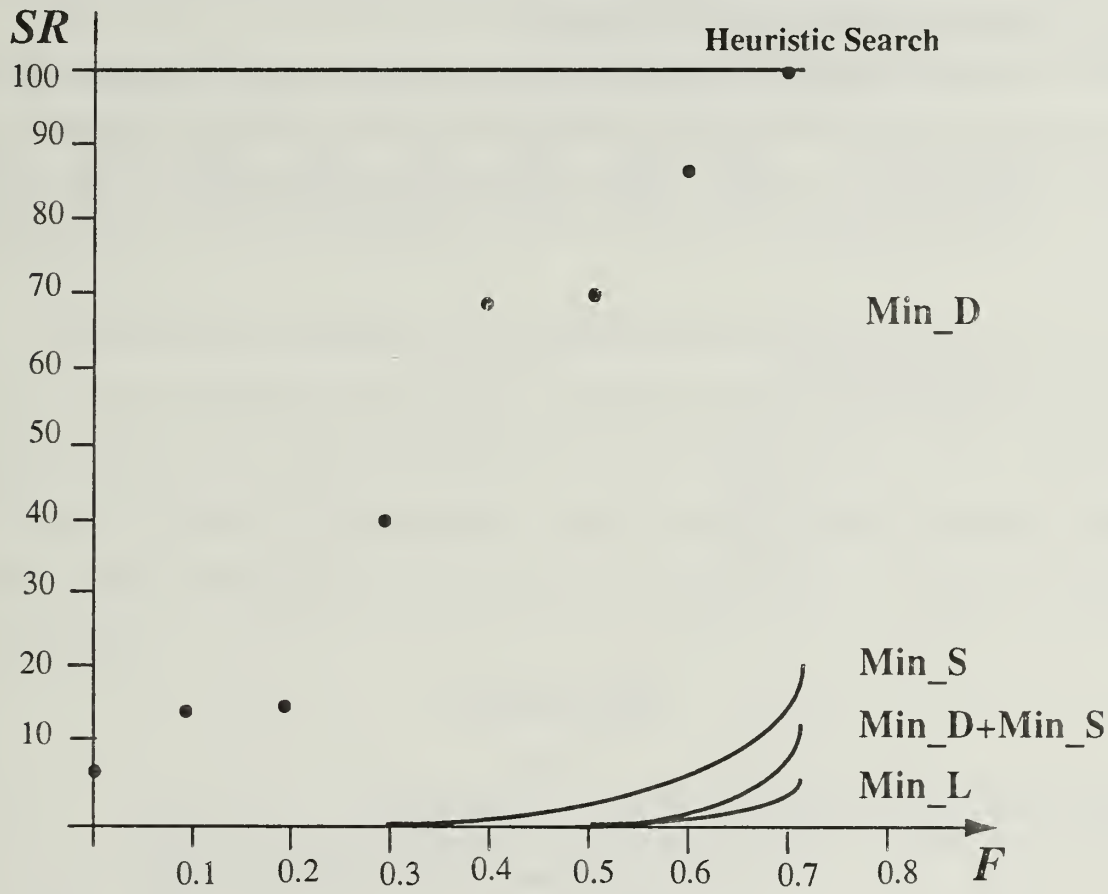

FIGURE 1. Relation between Success Ratio (SR) and Laxity (L)

The system, in our experiment, consists of three designers, one of each expertise level high, medium and low. Tasks durations are randomly chosen between Min_D (2) and Max_D (20). The number of task sets generated is 50, and each task set has between 28 and 31 tasks. We present the results as shown in Table 1, and in plot form in Figure 1 where the success ratio SR is plotted on the Y-axis and F on the X-axis (F is related

to laxity). Simulation parameter is F to measure the sensitivity of each heuristic algorithm to the change in laxities

As can be seen from Figure 1 the greedy heuristics Min_D, Min_S, Min_D+ Min_S and Min_L perform poorly due to the dependency relations between the tasks. We found that the heuristic search algorithm has a success ratio of 100% even when the deadlines are very tight (F=0). It is worth noting that this excellent performance by the heuristic search algorithm is obtained with unlimited backtracking. This leads us to study the effect of limiting the backtracking.

TABLE 2. OBSERVED BACKTRACKING (AS PERCENTAGE OF N) AND LAXITY (L)

Laxity (F)	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7
backtracking #	.57	.27	.16	.075	.034	.012	0.0	0.0

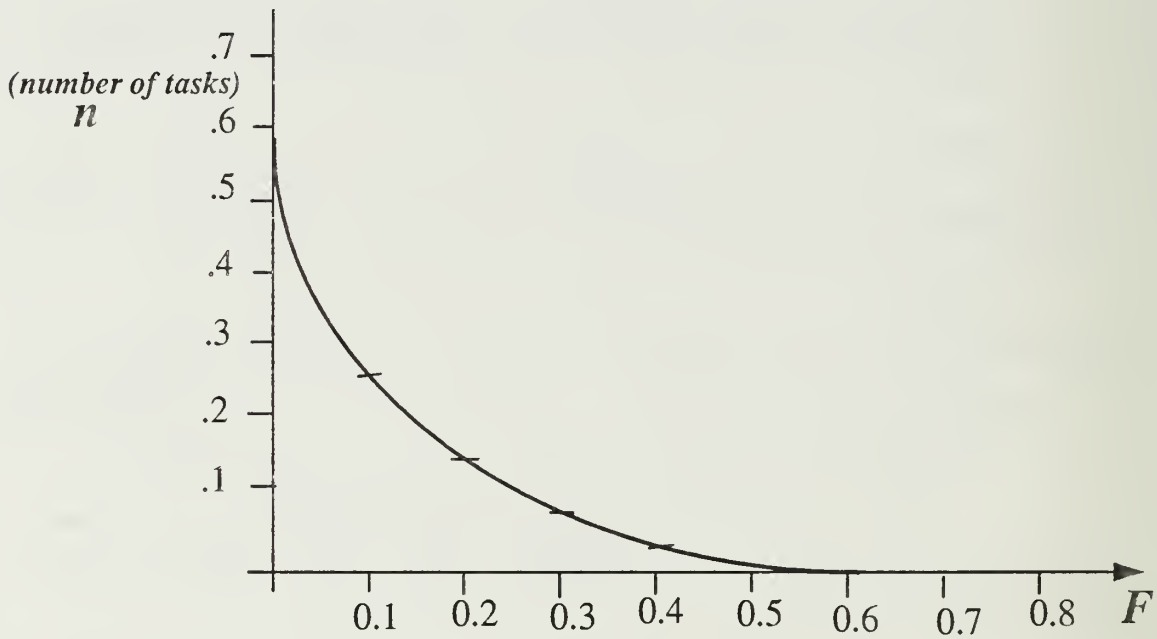

FIGURE 2. Limiting Backtracking

Instead of trying different backtracking limits and studying their effects on the performance of the algorithm, we do it the other way around by counting how many times the algorithm backtracks to get a feasible schedule given the different task sets. The results is shown in Table 2 where the number of backtracking is represented as a percentage

of the total number of tasks in a task set. The results plotted in Figure 2 shows that the backtracking limit in the worst case (tightest deadlines: F=0) is approximately 0.6 N, where N is the number of tasks in a task set, and this limit decreases significantly as the deadlines are relaxed.

E. CONCLUSION

The results of this study indicate that heuristic search can be a practical and very effective method for scheduling the tasks in a software development project. We also find that precedence constraints can have a significant impact on the relative performance of well known scheduling heuristics, and that the earliest deadline heuristic performs well in this context.

Our method has the advantage of providing constant monitoring of the status of the project to detect situations where deadlines must slip as early as possible, and providing suggested adjustments to project deadlines that reflect declared priorities when it ceases to be possible to meet the original deadlines. Such suggestions provide a feasible baseline schedule adjustment against which the project manager can evaluate alternative responses to the situation

REFERENCES

[1] Badr Salah, "A Model and Algorithms for a Software Evolution Control System", Ph. D. Dissertation, Computer Science Department, Naval Postgraduate School, December 1993.

[2] Hong K. and Leung J., "On-Line Scheduling of Real-Time Tasks" Real-Time Systems Workshop, May 1988.

[3] Ramamritham K., Stankovic J. A., Shiah P., "Efficient Scheduling Algorithm for Real-Time Multiprocessor Systems", COINS Technical Report 89-37, Dept. of Computer and Information Science, University of Massachusetts, 1989.

[4] Stankovic J. A., Ramamritham K., Shiah P., and Zhao W., "Real-Time Scheduling Algorithms for Multiprocessors", COINS Technical Report 89-47.

[5] Xu Jia., "On Satisfying Timing Constraints in Hard-Real-Time Systems", IEEE Transactions on Software Engineering, Vol. 19, No. 1, January 1993.

[6] Xu Jia., "Multiprocessor Scheduling of Processes with Release Times, Deadlines, Precedence, and Exclusion Relations", IEEE Transactions on Software Engineering, Vol. 19, No. 2, February 1993.

Distribution List

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Library, Code 52 Naval Postgraduate School Monterey, CA 93943	2
Research Office Code 08 Naval Postgraduate School Monterey, CA 93943	1
Dr. Luqi, Code CS Lq Naval Postgraduate School Computer Science Department Monterey, CA 93943-5118	10
Prof. Valdis Berzins, Code CSBe Naval Postgraduate School Computer Science Department Monterey, CA 93943-5118	30

DUDLEY KNOX LIBRARY
3 2768 00327445 7